# Testing in DevOps

eGuide

TECHWELL™

TECHWELL™

In order to fully realize the potential of DevOps and a team's ability to deliver higher-quality software faster and more securely, testing must be done continuously throughout the software development lifecycle. But continuous testing isn't just about automation and tools; there are cultural and process shifts that must happen, and everyone must be on the same page. This eGuide collects methodologies, processes, and tips to help your organization venture into continuous testing in DevOps.

# In this Testing in DevOps eGuide

# Why Software Testing Is Key to DevOps

*By Alan Crouch*

One of the major reasons organizations adopt DevOps practices is to accelerate delivery of software to production. This includes deploying more frequently and reducing lead time.

However, many organizations fail to include quality components in their practices. This leads to organizations delivering code faster, but unfortunately, that code is just poor. Continuous deployment without quality is just delivering continuous bugs to your customers.

If this sounds like your organization, software testing may be the missing component to your DevOps program. Top performing DevOps organizations, like Netflix, Amazon, and Etsy, utilize automated regression, performance, load, and security testing to ensure software quality is built into their DevOps pipelines and ensured by being forced to be run on every build. For Netflix, this allows software to be committed, tested, and fully deployed to production within sixteen minutes!

If your organization isn't as large or doesn't require as rapid deployments as those companies, automated testing in your DevOps pipelines still provides significant benefits. Even a small amount of automated regression testing can ensure basic tests are always run on a build. Typically, organizations start by building an automated smoke test. This smoke test provides a sanity check on a developer's code. This can not only reduce some of the overall manual testing effort by the testing team, but also ensure effort isn't being expended on builds that don't meet minimal quality standards.

More robust automated testing suites can result in even less manual testing, in addition to better-focused exploratory testing efforts into high-risk areas of the application, including interfaces, misuse cases, and the most important assets to your system. While not every test can (or should) be automated, spend your critical resources wisely by only focusing manual efforts on the things that really require their attention.

Most organizations today use some level of automation when looking at performance, load, and security. Leveraging those existing capabilities into your DevOps pipelines ensures that your deployments are not held back by independent groups late in the software lifecycle and they aren't an afterthought when the application is already in production. This has immediate benefits, such as reducing cyber security impacts on your applications and their data by ensuring critical vulnerabilities are identified earlier in your software development lifecycle and aren't deployed to production in error.

Pulling software testing into your DevOps practices doesn't have to be difficult. You can start by bringing your existing testers and security engineers into your planning sessions. Next, ensure each phase of your pipeline has a quality gate and software quality criteria that should be met in order to move to the next phase of your pipeline. Last, identify gaps between your quality goals and reality, and ensure you prioritize efforts into the activities that provide the greatest return on investment—those that are either run with the highest frequency, reduce the greatest software quality risk, or reduce the biggest bottlenecks.

These simple steps will lead to not just delivering code faster, but delivering better code faster.

# Test Everywhere: A Journey into DevOps and Continuous Testing

*By Anastasios Daskalopoulos*

In over ten years of leading the test and quality assurance processes for companies that did not have formal testing, one thing I have noticed is how difficult it is to persuade many developers and their managers that quality assurance standards and practices exist, and that the company would benefit from using these latest tools and methods.

It's a constant challenge to remove the concept that testing is a financial drain and time bottleneck that is a threat to product delivery. My counterargument has always been that post-release bug fixes from poorly tested (or even untested) software are more expensive and damaging to the company reputation, and properly planned and executed testing processes do not cause delays or excess costs.

Most of the developers at my new company had never worked with a full-time tester and had no knowledge of any software testing processes. "Testing" involved another developer on the team briefly looking at newly developed features before pronouncing them satisfactory for release to the client. Test plans, cases, reports, or any written test artifacts did not exist.

The reason for bringing me on as an independent QA person was to allow developers to concentrate on coding. Many developers had the impression that I was there to quickly review new websites and features before deployment to production. The initial idea the project managers and developers had for testing was to give me a few hours "to check if anything's broken" before giving the software to the client.

Although a step in the right direction, creating any test documentation was a gamble because the company had multiple teams that were simultaneously working on several projects, so it was difficult to predict the completion status of any software because of sudden and often undocumented changes. My pleas to be included earlier in the development process and develop automated regression suites were regarded as a good idea, but in practice they never proceeded beyond that point.

My presence in the company did improve software quality, but the "over the wall" philosophy still prevalent at the time did prevent more significant quality improvements. Luckily, new additions to the company and a switch to DevOps created an opportunity to shift the testing process far to the left.

So now I'm embarking on a new endeavor: introducing my company's teams of talented developers to the concept of continuous testing and implementing it within the DevOps framework.

## Planning for Quality

A fortunate convergence of a newly created DevOps team and newly hired lead developers were the main catalyst for a shift in our thinking about who was responsible for quality, at what point quality would be a concern, and what practices would be used to ensure quality from the inception of any project.

As the quality assurance specialist, I was tasked by the new DevOps team and new lead developers with helping to create a process for testing at each stage of development. My goal was to have testing and development work in parallel and act in conjunction for their mutual benefit.

Instead of the usual practice of banging out the code for a list of features for every sprint, a prioritized list of features is recorded in descending order in a TESTME.md file, a short, markdown-language file that uses the Gherkin syntax. This list focuses both the developer and tester on the simultaneous coding and test creation for each feature by importance.

*Post-release bug fixes from poorly tested (or even untested) software are more expensive and damaging to the company reputation.*

At this point, my job as the QA specialist is to concentrate on writing the project's test strategy, the sprint test plan (usually no more than two pages long), functional test cases, and test results.

## Unit and Integration Tests

Instead of developers relegating the task of "checking to see if anything's broken" to the tester, very early in the project they now prioritize the features within sprints, practice test-driven and behavior-driven development, and verify the functionality with unit testing and behavior-driven test automation frameworks.

Although not a written policy, many developers have seen the advantage of test-driven development and, on their own, have begun to create unit tests and write the code for the tests to pass.

The same is true for integration tests. The developers do not have a hard-dividing line on what defines a unit test versus an integration test; they generally agree to just write a test and define it as an integration test for organizational purposes if it involves the interaction between two independent classes.

After executing unit and integration tests, the developer now moves on to API tests, which are done using an automated tool, depending on the project.

## Functional Tests

By this point I've taken the time and opportunity to review the creative briefs, requirements, and design documents, as well as the TESTME.md list of prioritized features. Each sprint has also been planned and features for development and testing have been defined, giving me time to create tests for the sprint. I subject the tests to informal review to get more ideas about test priorities, general tips, and what to test in detail. This is an example of development and testing being done in parallel for the mutual benefit of each process.

*We have a faster testing process when the software is tested throughout the development lifecycle, by developers, testers, and automation alike. DevOps requires continuous testing, so we need a constant focus on quality.*

I also create end-to-end tests, such as for a user logging into a site, searching for a product and adding it to the cart, then going through the purchase process, or for searching through a large document library for a specific document and then downloading it. I review and execute the tests to gauge how well the newly developed features work with the existing features.

The test results are collected and made available as screenshots to everyone in the project, as well as to the client. An important point in testing is transparency, and value is added to the project when weak points and unstable areas have been found and fixed. The purpose of test results is to show initial unstable areas that have been fixed, what kind of issues were found, and that the issues no longer exist in the applications under test.

## Regression Test Suite

Once it is confirmed that the functional tests have passed, I create a regression set automated with a behavior-driven test tool. Experience has shown that software features that have previously passed and function reliably in production can fail after the inclusion of new software features, so tests that go into our automated regression sets confirm that features that previously passed in both manual and automated functional sets still pass. For example, passing manual and automated functional tests for a document library search will go into a regression set to verify that the search still passes.

At this point, all tests can be executed in the pipeline.

## The Changing Face of QA

One observation I have made is that throughout this process, my place as the QA specialist has changed. The initial role was at the bottom rung of the development process, where testing simply "checks if anything is broken," and now I'm floating above the process to verify that the correct quality processes are done by development.

Even in DevOps, testers should create and execute manual tests. Behavior-driven tests early in the process cannot catch all bugs by automation, since the stress is on feature behavior.

At my company, a focus on software quality now permeates every project. The changes we enacted have shown that we have a faster testing process when the software is tested throughout the development lifecycle, by developers, testers, and automation alike. DevOps requires continuous testing, so we need a constant focus on quality.

# Shifting Your Testing: When to Switch Gears

*By Maximilian Bauer*

Everyone keeps talking about shifting testing left. A few weeks ago, it was shifting right. But where are you now?

Before you shift your testing practices, it's important to evaluate your current position. Let's look at the benefits of shifting either left or right, including what needs can be met and what can be improved by shifting.

It can be helpful to have a visual, so let's use something already associated with shifting: a car's gearbox! As we're talking about testing, imagine the set of gears below.

## The Middle: Third Gear

This is the current situation for most companies. In development, only basic and technical checks are made. You have a dedicated testing system where your quality checks are done. In general, developers develop and do the technical stuff, and testers check the functions at different levels and in combination with other applications.

Apps are packaged so that the connections between them are dedicated to their individual functions and possibilities. This setup is rather static and requires much effort in maintenance and knowledge of each product, technology, and workflow. Every application is dedicated to a certain team, developer, or company, and those people take care about it. If you suffer a problem, you have to ask this one guy, and if he is not available, you are stuck. Information stored in human silos decreases your time to market significantly.

I worked with a bank that employed this setup. They had dedicated systems with straightforward testing in every stage. Test cases where done for dev or test systems, and real production problems needed to be recreated in test to reproduce them.

Problems included a long time to market, inconsistent test data through the systems, differences between the environment of each system and stage, and some errors occurring only when certain systems where connected. Most of the time testing went into the setup and then tried to align all systems and imagine them as they would run in production.

A database of test data that could just be imported was a help, because users and accounts could be created on the fly with any date on them; otherwise, testing would have taken ages. But is there a better way?

## Shifting Left: First Gear

Why do we test so late in our development lifecycle? If we manage to find bugs earlier, we can solve them more easily, not to mention more inexpensively. So let's shift testing left along the software development timeline, moving it into the dev environment.

This is possible because there was a shift in setup, too. Systems have migrated to the cloud, and testers have moved away from dedicated connections for every application to an API. In an environment controlled by an API layer, every app calls a function over a service, either a generic or dedicated one, enabling earlier testing. Checks are done over the services. Data is passed through a request and is received by the response. If a system is not available, service virtualization makes it reachable.

Service virtualization is just a way of mocking any kind of connected system, letting testers play around with it. The simulation can even be done with complex environments, and people can get training and practice like in the released setup, with much less risk. Data streams are recorded and will be replayed once needed. Service calls are made to the API layer and the virtualized service answers the same way a productive system would do. The application is part of the whole environment, and a huge landscape is covered already in development.

A module-based testing approach also changes the game. Developers create modules that are used later by testers. Just think of different perspectives on the system under test: Dev only looks at their part, while testers look at the combination of systems and see the bigger picture throughout the system landscape. Reusing some of the artifacts would create a new way of creating process chains or can be used for extending the simulation, which is just changing the

*Before you shift your testing practices, it's important to evaluate your current position.*

start and end. Developers can define connection details like users, passwords, endpoints, and security up front, and testers just take it for the test environment. (Yes, just a test environment. It is not meant to be a dedicated testing area without virtualization, where the systems really interact with each other.)

Test data and variants of all tests can be shared between dev and test, so a useful test can be run in dev already, always depending on the importance and the recorded data flow. Multiple people can share knowledge about the steering. The service call can be very generic and is highly dependent on the data provided. This means finding bugs earlier, which results in less testing effort in further stages and shortens your time to market.

I also worked with a credit card company in the process of shifting their testing left in this manner. They had managed already to get rid of the static staging and move to APIs, and even though they were not generic enough and still more specific to a certain function at this point, they still detected many errors early. They had a dedicated testing team just focusing on the API layer and checking the connection between it and the applications. Testers created modules with developers, representing the requests and responses. The same modules also could be reused in testing with different connection parameters, which saved a huge amount of time because everything was predefined, and the dev team knew what parameters the testers needed in case of any change. Big issues and data inconsistency were already checked in the dev environment and could be solved up front in the deployment to test.

They had just started to mock apps with service virtualization, but for the one they had done already, the developers told me how far their checks can go now because of the application behavior, which decreased maintenance and testing time later on.

## Shifting Right: Fifth Gear

This method is special because we kind of mix it up and try to get the best of both worlds. We increase our testers significantly, but our time to market needs to be very short—because testing is in production.

One of the best examples for shifting testing right is Amazon. Their customers and users of their product are the testers. Operations are highly integrated and get more significance in QA. This works best if you have already gone through shifting your testing left and have just one big product that moves through an automated build.

To ensure testing in production works, more effort is needed than in the other gears. Testing processes have to be defined close to perfectly and all work should be automated, favoring the API layer. Whatever you forget to check will be present in production, so an area that wasn't covered in test could lead to real damage. Your product should already be super stable and maybe not have too many things connected—or at least everything should be tested with above 90 percent risk coverage—or else operations would not be capable of doing a deployment.

Every part should be handled independently. One deployment to production should consist of small bits and pieces, because this ensures better test coverage. Huge features can cause more side effects than expected, so you want to have them included in your tests as early as possible. The final test itself will be handled through the users, and their feedback will provide you with the bugs.

Nevertheless, it is required to have a fully implemented continuous integration QA pipeline, which will be triggered during the build. Most of the tests happen automatically within the CI tool and the build is only done if tests are green. Ops will take care of the delivery pipeline, so testers need to create and maintain tests and check the

*Different testing strategies and approaches can be used for different stages, and if your process moves you to a different stage, then shift in that direction.*

results. In the best case, testers should be able to recognize false positives and assess the need for test or dev involvement.

If you walked through the whole gearbox, you should gain an advantage from the first gear. The API modules from dev could be reused here, and maintenance on a technical level is just done once. The final execution state is based on feedback. There is no need any more for go-live discussions after your tests; it's more about getting emails from upset customers.

It is a good option to increase user acceptance and make features based on user experience, but your product needs to be at a certain level to go for it. If your change does not affect too much of your business risk—meaning if both damage in case of failure and frequency of using it are low—you can easily trust your gear. In every other case, try to be sure that all tests cover as much as possible and all side effects are detected. The higher your identification and test coverage are, the higher the trust in your pipeline and your setup for delivering your product quickly and securely.

I'm currently setting up this process for a client that's a mobile provider. They have their application ready and are just adding features on features, so smaller parts. Every third-party application is virtualized and the connection runs through APIs. Every build passes through a CI pipeline and is checked on the dev side up front with automated API and JUnit tests.

If this is fine, it's moved automatically to a dedicated test environment. There, a CI tool reimagines the machines, and the solution, including environment and testing tool, is built. Once done, automated functional tests are run using UI, API, and real devices. If all tests are green, the build will be passed to production and built there, without any manual input. This requires trust in the systems and applications, but it saves time and money.

Changes in tests happen through all stages just by maintaining a single object. One test is created and affects multiple environments, thanks to prebuilt modules. The biggest challenge here is the alignment in the virtualization of the different stages. During test case creation, we introduced a staging and used the "four eyes" principle to check that everything was tested and covered.

## Don't Be Afraid to Switch Gears

Every environment is dependent on many factors, including application, team, and infrastructure setup. Different testing strategies and approaches can be used for different stages, and if your process moves you to a different stage—or if you think a different stage could serve you better—then shift in that direction.

Every shift is a process, not a moment; your journey will create the moments for you. Don't hesitate to shift your testing through the different phases of the software lifecycle if you think it will benefit your processes more. And don't let the idea of a having to create an entirely new test plan hold you back: Test reusability is high, and something defined in the third gear may save you time and money in the fifth.

Let your test cycles be your guide. Just like when driving a car with a manual transmission, if the engine starts to whine or you're afraid you're about to stall out, shifting gears may be just what you need.

# Continuous Testing, Shifting Left, and Test Automation: Getting It Straight

*By Wayne Ariola*

Today's executives face relentless pressure to deliver innovative software faster than competitors. Digital transformation has elevated software delivery to a C-level conversation, and software testing must be included in that conversation.

On one hand, testing is all too often the roadblock that stands between highly accelerated dev processes and highly automated ops-driven delivery processes. But on the other hand, testing is essential for ensuring that the release doesn't place the business at risk—undermining the very customer experience that digital transformation is dedicating to enhancing.

How can you achieve the optimal balance between speed and risk to deliver engaging customer experiences faster than competitors?

This is where continuous testing comes in.

Continuous testing is the process of executing automated tests as part of the software delivery pipeline in order to obtain feedback on the business risks associated with a software release as rapidly as possible. It really boils down to providing the right feedback to the right stakeholder at the right time.

For decades, testing was traditionally deferred until the end of the cycle. At that point, testers would provide all sorts of important feedback ... but nobody really wanted to hear it then. It was too late, and there was little the team could feasibly do except delay the release. With continuous testing, the focus is on providing actionable feedback to people who really care about it, at a time when they are truly prepared to act on it.

At some point the concept of continuous testing was inappropriately conflated with the "shift left" trend. But to deliver the right feedback to the right stakeholder at the right time, continuous testing needs to occur throughout the software delivery lifecycle—and even beyond that to production, like monitoring information from production and feeding that back into the quality perspective.

Just as the name indicates, continuous testing involves testing continuously. Simply starting and finishing testing earlier is not the goal.

So where does test automation fit in? It's essential for continuous testing, but it's not sufficient.

Test automation is designed to produce a set of pass/fail data points correlated to user stories or application requirements. Continuous testing, on the other hand, focuses on business risk and providing insight into whether the software can be released. Beyond test automation, continuous testing also involves practices such as aligning testing with your business risk, applying service virtualization and stateful test data management to stabilize testing for continuous integration, and performing exploratory testing to expose "big-block" issues early in each iteration.

When it comes to digital transformation, it's not simply a matter of more tools or different tools. It requires a deeper change across people and processes as well as technologies.

# Demystifying DevOps: A Day in the Life of a DevOps Tester

*By Justin Rohrman*

The idea of working as a test specialist on a team using DevOps can feel intimidating. There are at least two technology stacks, containerization and continuous integration, that you need to be familiar with. Add a source code repository like Git, a few test frameworks, and a scripting language to bundle everything together, and you start to approach a useful skill set.

My experience has been that very few people need to be able to start from scratch. Here is what a normal day of testing in DevOps looks like for me.

My current team follows pairing and Extreme Programming practices. For every code change, there are two developers and one test specialist. Most of the time, we start a change request by building a new test. This might be in RSpec or through Cucumber. Either a developer or a test specialist will write the test, and then a developer will write code to make that test pass. This helps us to understand the code we are writing, to refactor with less worry, and to know when we are done.

Throughout this red, green, refactor process, we are building new environments locally. One of us will push a change, and I might pull those changes and build a local environment.

The containers come in when we are closer to a usable version of the change. Many of the DevOps stories I read involve using commands I can never remember in workflows that feel archaic. I am able to do everything through our continuous integration system.

First I use a drop list to select the branch I want to use to build a new container. Building the container takes about five minutes. After my container is built, I select which environment to deploy to. After a few more minutes, I have a production-like environment available to explore and perform test ideas that are more complex than the programmatic testing we do during the development flow.

Pairing and having a test specialist removes most of the easy-to-find bugs that normal development leaves for the testing role. Generally I will still find problems related to the unpredictable way customers can use the product. I demo these bugs, then we make a decision as a group about whether they matter. Deciding to make a fix puts us back in the red, green, refactor cycle.

Over the course of a feature change, I might make one container, or I might make several. But it is very easy, taking somewhere around seven minutes each time I need to create a new environment from scratch. Once we feel like we are in a deliverable state, we create one last container environment in a branch merged with the head branch and demo to our product owner. That demo includes a discussion of any lingering things we decided not to fix so we can get some fresh perspective on them.

Testing in DevOps can feel scary, but like most things, once you get in a day-to-day groove, it becomes an indispensable part of your workflow.

# Why You Need Continuous Testing in DevOps

*By Tom Alexander*

The agile process is all about using short, flexible development cycles to respond quickly to customer needs. Doing this effectively these days involves building a DevOps software pipeline in order to quickly get high-quality software into the hands of your customers and receive feedback.

Most DevOps initiatives start with the adoption of continuous integration (CI) practices, where code is continuously integrated to make sure everything works together. Developers start the CI process by checking code into a shared repository many times a day. Each check-in is verified by an automated build process and some fast-running tests, allowing teams to detect errors and conflicts as soon as possible. Regression tests are run at least each night to make sure any changes made during the day did not break something else.

After CI is performed, a continuous delivery process is followed, where the application is further tested and, once it passes all the required tests, it's available to release into production. The upside of continuous deployment is that it delivers new functionality to users within minutes, as well as instant feedback to the team that allows rapid response to customer demands. Effective testing during your continuous deployment process is critical because without it, there is a big risk of continuously releasing buggy software into production.

## Don't Let Testing Practices Slow You Down

Continuous testing, which is often called shift-left testing, is an approach to software and system testing in which testing is performed earlier in the software lifecycle. The goals are finding defects earlier, increasing software quality, shortening long test cycles, and reducing the possibility of software defects making their way into production code during deployments. Continuous testing is critically important if your company is trying to use DevOps to deploy software frequently into production.

*Done right, continuous testing provides fast and continuous insight into the health of the latest build of your application.*

Done right, continuous testing provides fast and continuous insight into the health of the latest build of your application. This information can then be used to determine if the app is ready to progress through the delivery pipeline at any given time. Because testing begins early and is executed continuously, bugs are exposed soon after they are introduced, which reduces the time and effort needed to find and fix them. Consequently, it is possible to increase the speed and frequency at which bug-free, high-quality software is delivered, as well as decrease technical debt.

Technical debt refers to the price organizations pay when releasing badly designed code. It's a way of calculating the cost of additional rework caused by choosing an easy and quick solution now instead of using a better, less buggy approach that would take longer. Just like financial debt, technical debt incurs interest that must be paid, such as increased maintenance, support, or legal costs. By shortening the time it takes to fix buggy software, continuous testing helps pay down your technical debt by keeping these interest costs from accruing.

DevOps is a cultural shift that promotes collaboration among all teams, including development, quality assurance, operations, and others, such as performance management, release management, and maintenance teams. Consequently, there is no single product that can be considered the definitive DevOps tool. Often, a collection of tools from a variety of vendors is used in the stages of a DevOps process. Continuous integration is often seen as the backbone of a continuous delivery pipeline, which explains the popularity of CI tools such as Jenkins and Bamboo to build, test, and deploy applications automatically when requirements change.

Companies using DevOps often ship new software into production hundreds of times every day. These companies are delivering smaller pieces of software, collaborating, and monitoring in production to create a continuous flow of code, from check-in to production. And they're using continuous testing technology to weave testing activities into every part of their software design, development, and deployment processes.

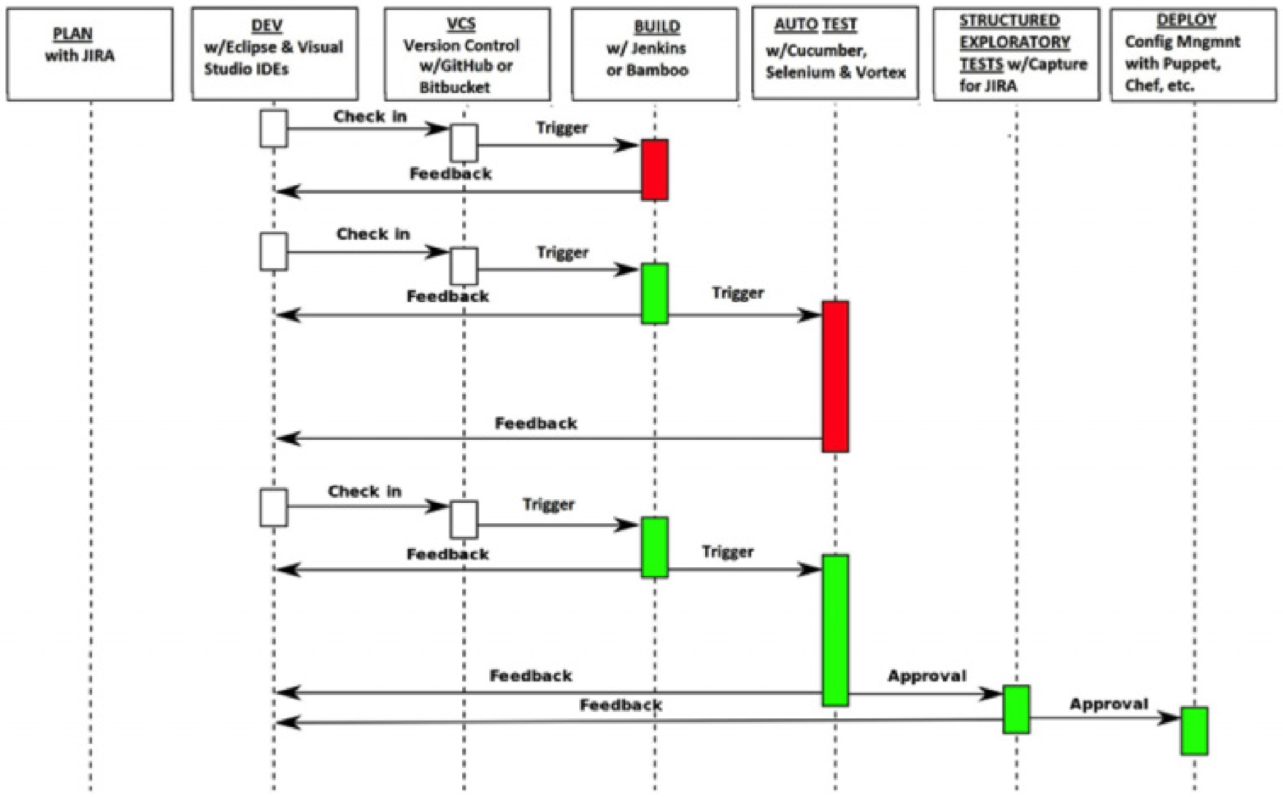
## Let Tech Do the Heavy Lifting

To release high-quality code faster, your organization needs to let tech do the heavy lifting by adopting next-generation tools and practices that enable you to test early, often, automatically, and continuously.

By executing the right set of tests at the right stage of the delivery pipeline—without creating a bottleneck—these tools enforce agile principles by providing appropriate feedback at every stage of the process. This enhanced communication averts duplication of efforts and increases alignment among dev, ops, and testing teams, which will allow you to deliver software on tighter schedules.

But these streamlined schedules are only possible if test automation is seamlessly integrated into your software delivery pipeline and DevOps toolchain. Test automation works by running a large number of tests repeatedly to make sure an application doesn't break whenever new changes are introduced. Manual testers are often still involved in DevOps projects, performing testing while an automation test suite is constantly running—but their role needs to shift toward a session-based exploratory testing approach, focused on areas with the most risk or where automation is not effective.

TECHWELL™


PLAN with JIRA
DEV w/Eclipse & Visual Studio IDEs
VCS Version Control w/GitHub or Bitbucket
BUILD w/ Jenkins or Bamboo
AUTO TEST w/Cucumber, Selenium & Vortex
STRUCTURED EXPLORATORY TESTS w/Capture for JIRA
DEPLOY Config Mngmnt with Puppet, Chef, etc.
Check in
Trigger
Feedback
Check in
Trigger
Feedback
Trigger
Feedback
Check in
Trigger
Feedback
Trigger
Feedback
Approval
Feedback
Approval

The image above shows an example DevOps pipeline that incorporates continuous testing during check-ins, continuous integration, and continuous delivery.

## A Continuous Testing DevOps Toolchain

While not an exhaustive list of all available DevOps products, here's a checklist of tools that together make up a viable continuous testing DevOps toolchain.

### Planning Tools

If you're looking for a tool that makes it easy for different teams to collaborate, Jira is an agile project management tool that supports any agile methodology, be it Scrum, kanban, or your own unique flavor. From agile dashboards to reports, you can plan, track, and manage all your agile software development projects. Jira's wide range of integrations also helps you connect to almost any other tool you're likely to need.

### Dev Tools: Desktop or Cloud-based IDEs

While Eclipse and Visual Studio are the most popular desktop IDEs, Cloud9, developed by Amazon Web Services, and JSFiddle lead in the cloud.

### Version Control Systems (VCS)

There are several web-based hosting services for DevOps version control, including Microsoft's GitHub, Atlassian's Bitbucket, and the open source GitLab service. All work within standard desktop or cloud IDEs to ease the processes of source code check-in and checkout.

### Build Tools

Jenkins is a CI/CD server that builds applications, runs tests automatically, and pushes code through your DevOps pipeline every time a developer checks new code into the source repository. Because of the rich ecosystem of plugins, Jenkins can be used to build, deploy, and automate almost any software project.

Bamboo is a CI/CD server from Atlassian. Like Jenkins and other CI/CD servers, Bamboo allows developers to automatically build, integrate, test, and deploy source code. Bamboo is a commercial software that is integrated and supported out of the box with other Atlassian products, such as Jira for project management and Hipchat for team communication.

### Automated Testing Tools

Cucumber is a tool for specifying application features and user scenarios in plain text. Cucumber runs automated acceptance tests written in a behavior-driven development (BDD) style that encourages collaboration on software projects by writing test cases in a natural language that nonprogrammers and domain experts can read. Selenium is a suite of different open source software tools that enable automated testing of web applications across various browsers and platforms. Most often used to create robust, browser-based regression automation suites and tests, Selenium, like Jenkins, has a rich repository of open source tools that are useful for different kinds of automation problems.

Agile teams can execute one-touch control of test automation from within the Zephyr platform with Vortex, Zephyr's advanced add-on that allows you to integrate with a growing suite of automated testing frameworks (including eggPlant, Cucumber, Selenium, UFT, and Tricentis) with minimal configuration. Besides being able to control the execution of thousands of automated test cases, Vortex makes it easy to automatically create test cases from test scripts and to apply insights from analytics on both automated and manual testing activities.

### Session-Based Exploratory Testing

PractiTest is a test management system that supports session-based exploratory testing practices. Session-based exploratory tests are created and added to a test set during testing. These tests can be combined with other types of tests, including structured manual and automated, to maintain test suites, traceability, and test coverage.

Capture for Jira helps testers on agile projects create and record exploratory and collaborative testing sessions, which are useful for planning, executing, and tracking manual or exploratory testing. Session-based test management, a type of structured exploratory testing, is an extremely powerful way of optimizing test coverage without incurring the costs associated with writing and maintaining test cases. Like Zephyr for Jira, Capture for Jira has a deep integration with the Jira platform, allowing users to capture screenshots within browsers, record screens in Chrome, create annotations, and validate application functionality within Jira.

### Deployment Tools

Longtime "movers and shakers" in the DevOps infrastructure-as-code space, Chef and Puppet are both automated configuration management and orchestration tools used to quickly spin up compute and storage instances on demand.

## Test Continuously to Deliver Faster

DevOps is more than adopting the right set of tools; it's a cultural shift that incorporates testing at each stage of the agile project lifecycle. Continuous testing is key to unlocking this culture change because it helps everyone involved communicate more, collaborate better, and innovate faster.

# Continuous Delivery Is Not a Pipeline

*By Clifford Berg*

"We have pipelines." That is what I always hear when working with organizations that claim to use DevOps and continuous delivery methods.

The claim resonates because pretty much every article you read about DevOps mentions "the pipeline." Graphic depictions of DevOps and continuous delivery almost always are a pipeline of some kind, showing a flow of software from development through various stages of testing and finally to release.

Continuous delivery is not really about the pipeline, however. In fact, in one instance, I worked with a team that had no pipeline but nevertheless delivered continuously, and I feel that the absence of a pipeline actually improved the behavior of the team.

I would claim, in fact, that the pipeline concept is a red herring. Continuous delivery is really about two things: testing strategy and branching strategy.

## We've Seen This Before

If you think about it, a pipeline is an awful lot like a waterfall process, just sped up. Worse, the pipeline job is really a reinvention of 1980s batch processing: You make some code changes, submit your job, and wait in line for it to execute so that you can obtain your results as a report (the pipeline tool's console log and the JUnit test report). Is that progress?

It is not. The only real difference is that today's pipeline doesn't take punched cards, and the output reports are accessible via a browser instead of a printout.

Consider what a team might have to do if they did not have a pipeline:

1. Deploy locally, or remotely using a script
2. Run integration tests locally, or remotely
3. Merge changes into the master, but after local integration tests pass
4. Deploy via script to a prod segment that receives a small percent of user traffic and gradually scale up

There's no mention of a build pipeline anywhere.

So what is the pipeline? Don't we need it?

Yes, but not in the way that it is usually portrayed. And organizations that use it the way it is usually portrayed are using it wrong.

The true role of the pipeline, which is manifest by a build orchestration tool such as Jenkins or Azure DevOps Services and the tests that it runs, is to run tests that cannot be run locally and to rerun all tests as a regression. It is a policeman: the tests are supposed to "stay green."

But if the team has the practice of running those same tests locally—or in isolation before they merge their code, which exposes their changes to other team members—then when merge does occur, all tests should pass. The pipeline would be green.

## Isolation Is Key

The key element, then, is running tests before merging. You don't need a pipeline to do that.

Notice also that in order to run tests before merging, you need a private place to deploy so that you can run your tests. That can be your laptop, or it can be a private area, such as a virtual machine in a cloud account. You must be able to deploy the system under test in a place where it will not replace the components that other team members are testing. In other words, you need to deploy in isolation.

Isolation is key for testing. Once isolated integration tests pass and you merge changes into the shared development branches, then—and only then—you are ready to deploy to a shared test environment. Thus, the real integration testing should happen before code changes reach the pipeline.

Some tests cannot be run locally, but they can still be run in isolation in a cloud account or data center cluster. Tests that often cannot be run locally include behavioral tests in a true production-like environment, network failure mode tests in which network anomalies are created, soak tests that run the application for a long time, and performance tests that stress the application.

The pipeline is a set of automated quality gates. However, if you are doing things right, you should have found most functional defects before code hits the pipeline. You do that by running some integration tests, and quality checks such as security scans, locally. This is known as *shift-left testing*, and it is how advanced DevOps organizations do things. If you are debugging functional errors in your pipeline, you are doing 1980s-era batch programming, and you are doing DevOps wrong.

*The pipeline concept is a red herring. Continuous delivery is really about two things: testing strategy and branching strategy.*

The pipeline is important; it is an integral part of DevOps. However, it is not the central element. The central element is the practice of testing continually using automated tests.

This enables programmers to have a "red-green" feedback loop in which they find defects as soon as possible—ideally, on their own workstation and before they merge their changes into the shared codebase—instead of downstream, where defects affect everyone else's changes and diagnosing problems is difficult.

The core to DevOps is the set of practices that make this shift-left approach possible. These include practices for branching and merging, as well as setting things up so that many kinds of integration testing can be performed locally on programmers' laptops or in cloud accounts that they have direct access to, so that a programmer can initiate an integration test run that occurs in isolation from all other programmers.

DevOps is a shift-left approach. The pipeline is important, but it is not the central paradigm.

# 5 Key Factors to Achieve Agile Testing in DevOps

*By Denise Rigoni*

In recent years, many organizations have been impacted by DevOps.

Some have implemented DevOps end to end by changing people's mindsets, automating deployment and build processes by implementing appropriate tools and processes, increasing test automation, breaking up silos between development and operations, and automating monitoring and reporting. Most organizations, though, have just started on their DevOps journey or are somewhere in the middle of the transformation process.

Part of the path to DevOps requires adoption of agile methodologies. What does it mean for testing when you switch from the traditional waterfall model, with a few long release cycles per year, to the agile model, with changes occurring every two weeks, or possibly even more quickly?

In most organizations, the world of legacy systems collides with the world of modern applications. However, the two need to coexist. In the past, we were executing regression tests for legacy systems over several weeks—manually as well as automated—but we don't have the same amount of time for testing anymore.

Switching from a traditional waterfall model to an agile one requires new testing approaches.

Optimizing test automation is an excellent way to bridge this gap. With automated tests in place, it is possible to allocate scarce testing resources to high-value activities, reduce time spent on test execution, and increase the number of test cycles possible in a shorter amount of time. The impact of these changes can be realized immediately with reduced efforts, cost savings, and dramatically improved time-to-market readiness.

My organization has implemented a lot of automation in the last few years—and made many mistakes in the process! But we also learned a lot and are now able to adapt better to a world that's becoming more and more agile. Automation only works if you have robust tests that can run unattended from the beginning to end.

*Switching from a traditional waterfall model to an agile one requires new testing approaches. Optimizing test automation is an excellent way to bridge this gap.*

Here are five key factors to achieve agile software testing in DevOps.

## 1. Test data management

Having the right test data is the first important step in automation. If your test data is not stable, you will never succeed. According to software industry statistics, nearly 30 percent of test execution failures are due to improper test data.

For each of your test cases, define what data is needed in order to execute it. Keep it flexible, describing the attributes your test data needs to fulfill rather than the test data itself. For example, "John Doe from UK" is the test data itself, and the correct definition would be "Male natural person with nationality UK."

Once you have defined the required attributes, where is the best place to find test data? To be completely flexbile, and with the new GDPR legislation especially, it makes sense to build up required test data synthetically. This is a good option if you don't need historical data (even though that is also possible to build) and if your data setup is not too complicated or distributed across too many different systems.

If you require more complex data, you might use existing customer data that has been anonymized. In this case, queries are needed to find data in the databases matching the exact combinations of attributes.

Usually, a combination of both approaches works best. Creating synthetic test data and running queries on a database to attain highly suitable test data can and should be automated.

Proper test data management is not only a must for automation; it's required for manual testing as well. According to some surveys and statistics, manual testers spend 50 percent to 75 percent of their effort on finding and preparing appropriate test data. Test data must be independent from people maintaining it for you to get a return on investment.

## 2. Flexible steering

When you design your test cases, use steering parameters instead of copying the same test case several times to fulfill certain criteria.

Let's assume you need to run your test case first in a development environment and later on in a production-like environment. You would need to log in with role X and role Y. It's inefficient to copy the test case four times for each combination of environment and role when you can enter these parameters only once in the test case as a steering element. This reduces the overall maintenance of this test case by 75 percent.

Whenever possible, it's advisable to work with reusable test step blocks. Frequently and repetitively used test steps can be defined once and reused in different test cases. For example, with a login for an application or generation of a new client, which are used over and over again for subsequent test cases, there will no longer be the need for repetition, which saves valuable time and effort.

A failing test case often results in an end to the automated execution; therefore, the next test case cannot run, because the application was left in an "undefined" state. Imagine starting the automated execution in the evening to run the test cases overnight. The next morning you would find your test cases were not executed because the first test case that had failed and prevented all the others from running.

This can be avoided by defining recovery scenarios for each test case or making them completely independent of each other. Instruct the test case what to do in case of failure.

## 3. Test environment and service virtualization

Next to missing test data, unstable, unavailable, or incomplete test environments are one of the biggest time-consumers in testing.

Consider this scenario: Your tests are planned, testers organized and blocked their time on a certain day, and, finally, when you want to start testing the application, it is not available. Or the application itself is available, but another dependent application or dependent service delivering the data you needed is not.

The more people and applications involved, the more complex the test environment becomes, and the risk of unavailability increases.

However, in our modern agile world, the days of waiting to test everything until all components are developed and all systems and services are completed are gone. Instead, you need to simulate those components or systems that are not yet connected or developed, so testing can be started earlier. This is possible with service virtualization. You need to know the parameters a service expects and the data it returns.

A good service virtualization testing tool can react to changing circumstances and switch from virtualized services to the real service if it becomes available, then back to virtualized services if the service is not available for whatever reason.

Bear in mind you can't do all your testing based on virtualized services; at a certain point, you'll have to start real, end-to-end testing. But many critical defects will be found before your end-to-end tests if you start testing much earlier. This saves a significant amount of rework and debugging costs later.

Additionally, a well-managed test environment helps to improve stability. This can be achieved with organizational measures (e.g., a centralized test environment management team that coordinates and controls the test environments) or defined test environment management processes.

Many organizations do not want too many processes put in place because they believe agile development doesn't need processes. However, certain processes—or, rather, rules of engagement—are necessary to improve test environment stability.

These processes don't need to be very time-consuming, at least for more advanced test stages, such as end-to-end tests or business verifications:

- If several applications and services from different teams are involved, define time slots when deployments should happen—testers will then know when everyone is done and they can start testing
- Actions like restarts of applications and services, database refreshes, or other outages should be planned during time slots where no testers will test
- If testers communicate when they plan to test—either manual or automated test cases—such as by calendar, a test environment management team can better coordinate and communicate
- The more detailed new changes are communicated to the testers, the better they can adapt their test cases
- If scripted properly, DevOps automation can also help manage environments.

## 4. Test users

When automating test cases, it's best to never include your personal credentials in the automation. If you're out sick, your colleague won't be able to execute the test cases you've designed. Also, using another person's personal user account is typically a security breach in most companies.

The correct approach is to set up special test users that cover the different roles you require for testing your applications. Test users will persist if someone leaves the company or changes their role. Just make sure you take into account roadblocks like single sign-on, remote desktop connections, firewalls, cross-border access, or other barriers.

## 5. Continuous testing and continuous integration

Integrate your automated test cases into the continuous software development cycle. If the test cases are stable, why not run them every night or after every build to quickly get results about the status of the software? Ensure long-running tests are only integrated in test automation that runs during night builds, when enough time is available to finish execution before the next build starts.

Some automation tools integrate with standard development tools that provide continuous integration capabilities. Use distributed execution to run automated test cases on several machines in parallel to reduce testing time and get the results as early as possible.

> *Many organizations do not want too many processes put in place because they believe agile development doesn't need processes. However, certain processes are necessary to improve test environment stability.*

TECHWELL™

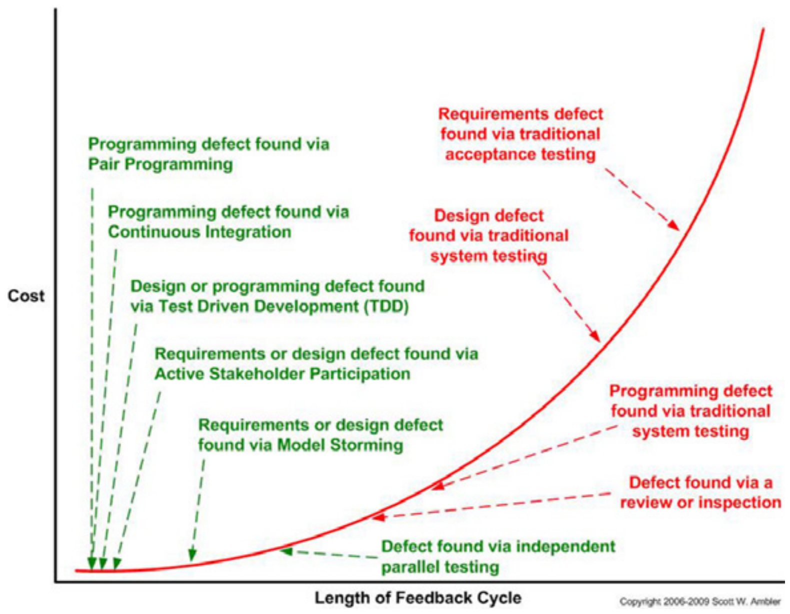
# Key Enablers for Continuous Testing

*By Max Saperstone*

Over the last year, I've heard a lot of buzz around the term "continuous testing," or testing before, during, and after each software change is made. This is both exciting and confusing for me: It's great that people are pushing for more testing, but isn't continuous testing something that testers have always been striving for? Still, any excuse for getting people excited about additional testing is a win. There are several activities your organization should adopt to move toward continuous testing:

- Collaborating on requirements (behavior-driven development, or BDD)
- Constantly validating change (regression testing)
- Dev-test pairing in exploratory testing and reviewing test cases
- Automated testing in CI (unit testing, API testing, and code analysis)
- Continuously improving test approaches, test suites, and test scripts
- Reviewing customer feedback and product ratings

There are also some enablers your organization should embrace. Let's look at three practices that will help your company get ready for continuous testing.



*You can't test continuously if you don't start testing early!*

## Whole-Team Quality

It's important to realize that QA does not "own" quality. QA doesn't introduce issues that impact quality, and they don't fix these issues. They simply find them and point them out.

Many of these issues could be identified earlier. If we shift the focus of testing to start sooner in the development process, we can save time and money.

Good quality code starts with the design of the software. Get developers to do test-driven development. Better yet, get the organization to adopt BDD practices: Have the developers, testers, and product owner all agree on functionality and tests before code is written. Shifting testing left is key to your continuous testing journey.

## Build Orchestration and Continuous Integration

Having an automated build and continuous integration (CI) pipeline is vital to being successful in your continuous testing effort. This way, once code is committed to a source control repository, testing can begin. This means you need to ensure that your application builds, unit tests pass, code coverage is at an acceptable level, and so on.

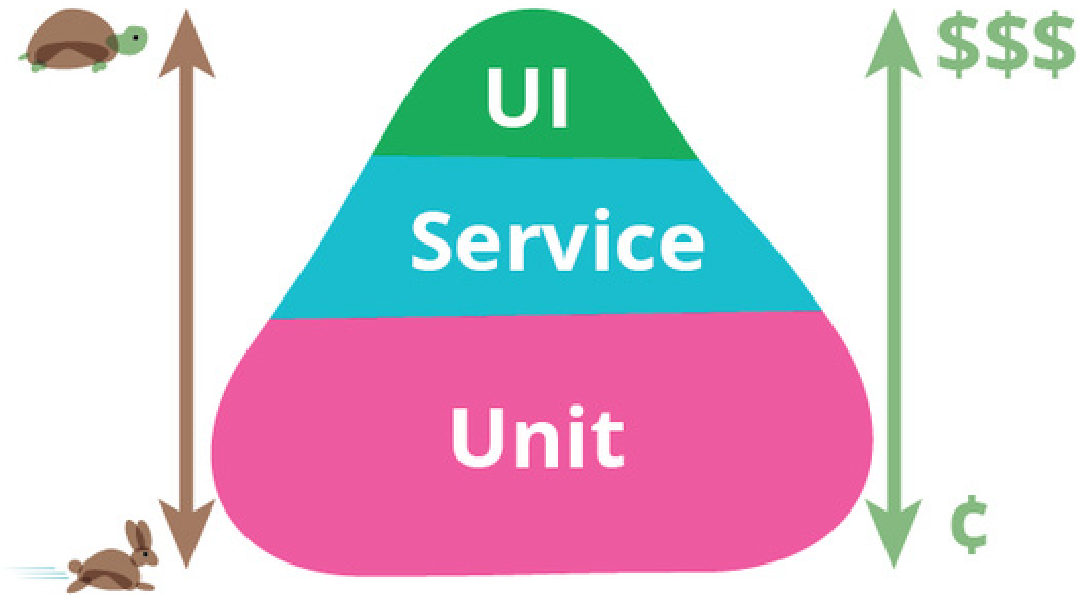
Developer-level testing is crucial, as it provides rapid feedback and ensures that at the lowest level, the code does what is expected. Consider what other things you can throw into your CI: static or dynamic analysis, some security checks, or maybe even some API validation. The more information, the better, as long as it provides rapid feedback to the team.

## Automated Testing below the UI

One big thing to remember when testing is that your application is more than a user interface. You probably have some APIs, a back end, and a database, all of which should be tested. Not only is it easier to test at these lower levels, but these areas can typically be tested earlier in the development process because they usually don't need to be fully developed in order to start testing them. This means you can get started testing sooner, which is of course the goal.

Additionally, API tests typically run tens to hundreds of times faster than the UI tests. This plays in perfectly with our idea of rapid feedback, and keeps pushing testing further left.

Remember, you can't test continuously if you don't start testing early!



TECHWELL™

# Insight from the Industry

*"In shifting left, teams see huge benefits such as finding bugs much earlier, sometimes even before code is ever written. This results in a higher quality of code and a far less expensive way in achieving such."*

—Angie Jones

READ THE FULL INTERVIEW

*"We're not going faster, we're not coding any faster, we're not typing any faster. We're learning good ways to slice these features down, so we can deliver small increments."*

—Lisa Crispin

LISTEN TO THE FULL INTERVIEW

**GAIN MORE INSIGHT WITH THESE VIDEO INTERVIEWS**

- DevOps Misconceptions and Testing Confidence: An Interview with Hans Buwalda
- Understanding the Role of QA in DevOps: An Interview with Gene Gotimer
- Shift Your Mindset with Your Testing: An Interview with Isabel Evans
- Deciding if You're Technically Ready for DevOps: An Interview with Sunil Sehgal
- Continuous Testing and Open Source Integration: An Interview with Alex Martins

TECHWELL™

# Additional Resources

## MORE INFORMATION FOR SOFTWARE PROFESSIONALS

STICKYMINDS™ A TECHWELL COMMUNITY

StickyMinds is home to thousands of software testing resources, including informative articles about all aspects of testing and interviews with industry notables. StickyMinds offers how-to advice and views on the latest ideas and practices from experienced software professionals and thought leaders. Join the community to gain access to exclusive members-only content such as conference presentations, Q&A discussions, a weekly newsletter, and more.

JOIN HERE

### NARROW YOUR SEARCH TO A SPECIFIC TYPE OF RESOURCE:

### StickyMinds Articles

StickyMinds articles cover a wide range of software testing topics, including testing in DevOps, continuous testing, test automation, agile testing, test process improvement, and much more. **Click here** to read articles about continuous testing and DevOps on StickyMinds.

### Interviews

Each year, TechWell interviews dozens of software professionals, including well-known thought leaders, seasoned practitioners, and respected conference speakers. **Click here** to read, listen to, and watch interviews with testing experts about continuous testing and DevOps.

### TechWell Insights

Read stories about continuous testing, testing in DevOps, agile, teams, and more, all written by software industry professionals. New stories are added each weekday, so **click here** to read the latest—or sign up to receive the weekly newsletter roundup of the newest stories.

### TechWell Conference Presentations

Couldn't make it to a TechWell conference to sharpen your testing knowledge? TechWell conference presentations are available to StickyMinds members for free soon after conferences end. **Click here** to join StickyMinds and access conference presentations related to testing in DevOps and continuous testing.

### TechWell Conferences

TechWell conferences feature keynote presentations, tutorials, and classes covering continuous testing, DevOps, agile, test automation, mobile testing, test management, and more. Learn from experts in the field and network with your peers to get the most immersive conference experience possible. **Click here** to learn more.

### The TechWell Hub

Join the active and engaging TechWell Hub community on Slack to keep in touch with what's happening in the industry and ask questions about your day-to-day challenges. The Hub offers a safe space to get real-time answers from other software professionals in your shoes, and even experts in the field! **Click here** to join for free.

coveros

**Accelerate Delivery**

Our partner, Coveros, has significant DevOps experience and can help organizations implement continuous testing and integrate automated testing solutions into their CI/CD pipeline. Coveros offers courses on automation, DevOps and DevSecOps, common development and automation tools, and fundamental and specialized software testing topics—all of which include best practices taught by industry leaders. Whether you're looking to get hands-on experience for yourself, your team, or your organization, Coveros has a learning solution for you.

**Automation courses** | **DevOps courses** | **Dev & Testing Tools courses** | **Agile Testing services**